3. Basic Features of Real Variables in the CREATE Step

3.1 Introduction

Chapter 2 illustrated the general purpose of the CREATE step: to prepare a VPLX tally file of replicate totals from an incoming file of individual observations. The resulting VPLX file is the essential connection to the rest of the system. For example, the DISPLAY step can display estimates, standard errors, covariances, correlations, and t-tests based on the VPLX file. Alternatively, the TRANSFORM step can derive new variables that are functions of sample totals and include them on a new VPLX file, which in turn can be input to DISPLAY. Every VPLX file is either a direct output from a CREATE step or is a descendent of a file from a CREATE step.

Most of the commands in the CREATE step focus on the treatment of the individual observations. For example, such statements specify how VPLX is to read observations from data files, recode some variables according to specified conditions, assign constant values to some variables, or treat some variables as categorical. A few commands in the step instead specify the contents of the outgoing VPLX file. Examples of both types of statements appear in this chapter.

This chapter presents some of the basic features of the CREATE step for real variables. The selection attempts to identify the subset of most frequently used commands. Subsequent chapters introduce other types of variables -- real with missing, categorical, crossed real, crossed categorical, and class -- that may also appear in the CREATE step. Other chapters describe additional features of the CREATE step, such as logical operators and declaration of scratch files.

The treatment of real variables in VPLX generally parallels analogues in languages such as FORTRAN and SAS. For the benefit of those familiar with the SAS language, Appendix B, Section B.2, translates VPLX commands into their SAS equivalents or near equivalents.

Chapter 4 describes how real variables created by the commands in this chapter may be displayed. Chapters 3 and 4 are essential reading as a background for almost any VPLX application.

Summary of this chapter:

- Section 3.2 describes the file specifications in the CREATE statement.

Sections 3.3-3.11 describe and illustrate operations to observation-level data:

- Section 3.3 introduces essential statements, INPUT and FORMAT, to read data from a standard character data file.

- Section 3.4 covers the CONSTANT statement used to assign constant values to variables.

- COPY in Section 3.5 copies the value of one VPLX variable into another.

- IF-THEN-ELSE in Section 3.6 permits conditional execution of many CREATE statements.

- Section 3.7 describes the elementary arithmetic operators ADD, SUBTRACT, MULTIPLY, and DIVIDE.

- PRINT in Section 3.8 displays the values of variables at the observation level.

- Section 3.9 illustrates the features introduced in Sections 3.2-3.8.

- LINK in Section 3.10 names additional input files with information that may be combined to form the observation. Each LINK statement then may be followed by its own INPUT and FORMAT statement. Conditional linkage, one-to-one, and one-to-many linkages are possible.

- Section 3.11 illustrates LINKing.

Sections 3.12 - 3.15 concern statements affecting the contents of the resulting VPLX file:

- SELECT in Section 3.12 allows the user to determine which observations are to be added into the overall estimates.

- Section 3.13 describes the WEIGHT statement, which allows the identification of a weight variable, the reserved variable name `weight`, and related statements.

- Section 3.14 introduces DROP and KEEP, which may be used to specify the variables to be included in the output VPLX file.

- LABEL in Section 3.15 provides labels for use in DISPLAY.

3.2 CREATE Statement

3.2.1 General Form. A line with CREATE starting in column 1 signals the beginning of a CREATE step. The statement must also specify two files. One file is an existing character input file, containing numeric data, to be read by VPLX, and the other is the output VPLX file. The program will terminate in error if the input file does not exist, but the output file may be either a new or existing file. If the output file exists, it will be overwritten.

As is the case generally with file specifications for VPLX steps, the output file must be different from the input file.

The syntax is:

```
  CREATE   IN =  fname1  OUT = fname2
```

The length of each file name is limited to 80 characters. The form of ***fname1*** and ***fname2*** depends on the host system. In many environments, either specification may be the full name or partially shortened name (depending on default directories, *etc.*) of the file.

Sections 10.2-10.3 detail the specification of files for VPLX, including system-dependent features. A few examples are given here, however, as a guide.

In a PC DOS environment, the CREATE statement might take the form:

```
  create  in = c:\survey\datafile.dat
          out = c:\survey\datafile.vpl
```

or, if c:\survey is the current directory, simply

```
  create  in = datafile.dat   out = datafile.vpl
```

As long as full names are specified, however, it is not necessary for the two files to be in the same directory or DOS drive. For example, the input file may reside on a CD-ROM drive, `d:`,while the output file must be directed to a drive for which the user has write access, such as a standard hard disk, `c:`. The .vpl extension on the file name is recommended as a way of identifying VPLX files, but this convention is optional.

Naming of files under UNIX is similar, although not identical, to DOS. (For example, UNIX employs " / " instead of " \ " in naming subdirectories.)

```
create  in = /data5/datafile.dat
        out = /tmp/datafile.vpl
```

UNIX distinguishes between lower and upper case in naming files, so that file names must be spelled in the correct case, unlike DOS. In most other respects, however, VPLX is case-insensitive.

In a VAX VMS environment,

```
create  in  = user$:[user_name.survey]datafile.dat
          out = temp$:[user_name]datafile.vpl
```

illustrates the naming of files. Default directories or shortened names may be used.

In some IBM environments under OS or CMS, it is necessary to use forms such as:

```
CREATE  IN =  VPLIN  OUT = VPLOUT
```

where `VPLIN` and `VPLOUT` are externally defined DDNAMEs of no more than 7 characters, requiring a DD statement in OS and a FILEDEF in CMS. It is also possible, in some versions of IBM FORTRAN, to reference the DSNAME directly. Extensions of VPLX to OS and CMS that would allow specification of file information, such as VOL=SER= *etc.*, are currently under development.

3.2.2 File Extensions. Note that in DOS, UNIX, and VMS, among other systems, the last part of a file name is often called the extension and used to distinguish types of files. VPLX does not enforce any particular system of file extensions. On the other hand, the author finds that consistent use of a convention is quite helpful. The author's own preferences are:

.DAT	for a character (e.g., ASCII) data set.
.VPL	for VPLX files output from VPLX.
.CRD	for files of commands to VPLX (*1*).
.VSK	for files containing VPLX code (similar to .CRD) with unresolved substitutions requiring the SET feature of VPLX (Section 10.10). These files are analogous to macros in some other languages. In a production system, a .CRD file may contain SET statements to define the substitution strings for the particular run, and then reference one or more .VSK files through INCLUDE (Section 10.5).

.LIS for output "listings" from VPLX. These files could be sent to a printer, although the author typically reads these files with an editor and actually commits a only few to paper.

3.3 Essential Statements: INPUT and FORMAT

3.3.1 INPUT. Currently, the INPUT statement is a required part of the CREATE specification. The function of this statement is to identify the variables to be read from a character data file to form an observation.

After INPUT in positions 1-5 of the statement, a list of variable names to be read from the input file must follow. Rules on variable names follow in Section 3.3.2. Chapter 2 presented a number of examples of the INPUT statement, including:

```
input    rooms persons cluster
```

Variable names may appear on as many lines as necessary, as long as position 1 is left blank on all continuation lines. (Virtually all VPLX statements may be continued in this way, exceptions, such as ECHO OFF, are noted in the descriptions of the corresponding commands (*2*)) At the end of the variable list, VPLX provides a count of the number of variables, which is sometimes useful for checking.

All variables on the input list become initially defined as real variables. Subsequent statements in the CREATE step can change their type or create new variables of different types from them.

3.3.2 INPUT Options: An optional but helpful feature requests VPLX to print observations at the beginning of the file for purposes of assuring that the data are correctly read. For example:

```
input    rooms persons cluster / option nprint = 3
```

will print the first 3 observations read from the file. In general, `nprint` may be set to any positive number, although usually printing a few observations provides an adequate check.

The actual printing occurs as VPLX begins to carry out the CREATE step. Consequently, the resulting printing of input observations does not generally immediately follow the INPUT statement in the listing, but instead follows the listing of all commands associated with the CREATE step.

Another option directs VPLX to stop reading the input file after a specified number of observations have been read. This feature is occasionally useful for debugging VPLX applications. The syntax is,

```
input    rooms persons cluster / option nobs = 4
```

for example. Both `nprint` and `nobs` may be specified together in either order:

```
input    rooms persons cluster / options nprint = 3
           nobs = 4
```

Note that `option` and `options` are both accepted spellings.

3.3.3 Kinds of Data That VPLX Reads. In general, the INPUT and FORMAT features of VPLX are among the most FORTRAN-constrained features of the system. VPLX reads only one type of data, real numbers. Integers may be read in this form, that is, VPLX has no difficulty reading example 1. in Exhibit 3.1 with the format (F1.0).

```
example    positions on record
           00000000011111111112
           12345678901234567890

1.         1
2.         1.
3.         27
4.         2.7
5.         .27000D+02
6.         .27000E+02
7.

BUT NOT

8.         .
9.         Y
```

Exhibit 3.1 Numbered examples of types of data that VPLX can and cannot read. Position numbers 1-20 are indicated above the examples by the 2-line numbering scheme. Example 7 will be read as 0. Because of FORTRAN standards, VPLX cannot reliably read the solitary point illustrated by example 8 with floating-point edit descriptors in the FORTRAN format. Except for the special meaning of "D" or "E" as they appear in examples 5 and 6, VPLX cannot read characters such as those illustrated by example 9.

VPLX is unable to read character strings of alphabetic characters, e.g. "`Y`" in example 9., "`CPS`", *etc*. It is, however, able to interpret examples 5. and 6., which represent numbers in scientific notation.

FORTRAN, and therefore VPLX, reads fields that are entirely blank, such as example 7., as "0" when reading real numbers. This is different from SAS, which generally treats blanks as missing.

SAS reads fields containing only "." as a missing value, such as example 8., whereas FORTRAN generally does not tolerate these values, although the treatment may vary somewhat from one FORTRAN environment to another. Consequently, fields of "." alone must be avoided on input to VPLX. (Of course, a decimal point included as part of a number, such as examples 2. and 4., is perfectly normal to FORTRAN.)

3.3.4 FORMAT. A separate FORMAT statement must be used to specify the FORTRAN format of the incoming variables. The format should describe the layout of the entire observation, corresponding to the entire list of variables on the INPUT statement, which may extend over more than one record. The primary edit descriptors are "F" and "D" format for input variables, "X" to denote skipped spaces, " / " to read the next record to continue processing the same observation, and " (" and ") " to bound the format and segments of description to be used multiple times. Each of these is described more below. Except to change "f", "d", and "x" to upper case, and to remove blanks, VPLX does not edit or scan the format, and relies on the FORTRAN implementation to use the format in reading the input file. The format should specify as many fields as the number of variables in the INPUT statement.

The format may be continued onto additional lines, if necessary. VPLX will squeeze out extraneous blanks, but the total length of the format may not exceed an upper limit depending on the installation. For the PC version, this limit is 2560 non-blank characters (*3*).

Unless there are linked files, as described in Section 3.9, the FORMAT statement may appear anywhere in the CREATE statement. For purposes of checking, however, it is good practice to place the FORMAT near the INPUT statement.

3.3.5 FORTRAN Edit Descriptors. FORTRAN edit descriptors describe the contents of positions on one or more records. The primary forms for VPLX are:

F*w.d* - For a right-justified floating-point number. The width, *w*, is a positive integer (>0) specifying the total number of positions occupied by the number, and *d* is a non-negative integer ($0 \le d \le w$) specifying the number of places after the decimal. If a decimal point is present, then it automatically overrides *d*, that is, the number will be correctly read regardless of *d*. Without a decimal, if *d*=0, then the number will be read as the floating point representation of an integer.

For example, F1.0 may be used to read a number in the range 0-9 from a single position. In Exhibit 3.1, F1.0 translates example 1. into the number 1. F2.0 translates the two characters "27" in example 3. into the number 27, but F2.1 translates example 3. into 2.7 and F2.2 into .27.

The decimal present in example 4. overrides any distinction between F3.0, F3.1, F3.2, or F3.3; thus, 2.7 is read in all cases.

Although leading zeroes are completely unnecessary before a decimal point, leading zeroes after an implied decimal point help to assure that the value is correctly read. For example, the number 4 will be correctly read from " 4" by F3.0, but reading " 4" with F3.3 may produce results that vary by FORTRAN implementation, and it is better to avoid this situation. (A safe alternative in this case is to read the number with F3.0 and then divide it by 1000 using the DIVIDE statement in Section 3.8.)

Warning: Left-justified numbers, with trailing blanks in the range covered by *w*, pose potential problems for some FORTRAN implementations. For example, all FORTRAN implementations read " 4" with F3.0 as 4., but their reading of "4 " is less predictable. For safety, users should always right-justify variables. (Users who employ SAS to manipulate their data for input to VPLX can accidentally produce left-justified data by using character variables. The situation is safer with real (numeric) items, however, since SAS right-justifies real (numeric) data output with a SAS PUT statement from a DATA step.)

D*w*.*d* - For right-justified scientific notation in FORTRAN. The number is expressed as a fraction followed by a power of 10. The width, *w*, is a positive integer (>0) specifying the total number of positions occupied by the number, and *d* is a non-negative integer ($0 \leq d < w$) specifying the number of places after the decimal for the fraction, although it is advisable to always explicitly include the decimal point. For example, " .200D+01" is read by D9.3 as 2. Both examples 5. and 6. in Exhibit 3.1 are read by D10.5 as 27. This descriptor is infrequently used with survey data, but is convenient for covariance matrices.

*w*X - To skip *w* positions. This may be used to skip over blanks, data not referenced by INPUT, and alphabetic data that VPLX cannot directly read. For example

```
input    psu  segment
format  (f3.0,3x,f4.0)
```

would skip over positions 4-6, which could contain "CPS" or other data that VPLX cannot read directly.

/ - To skip to the next record in the observation. For example:

1) If an observation is on a single record, " / " should not be used.

2) If the observation is on two records, " / " should be used once to indicate the point at which reading should begin from the second record. If the information is only on the second record, " / " should precede other descriptors to indicate that FORTRAN should go to the second record immediately. For example

```
input    psu  segment weight repw1 - repw48
format  (/,f3.0,3x,f4.0,49f8.2)
```

skips the first of two records and goes immediately to the second. If the information is instead only on the first record, the format should end with " , /)"

```
input    psu  segment weight repw1 - repw48
format  (f3.0,3x,f4.0,49f8.2,/)
```

to indicate that the second record should be skipped over entirely.

3) If the observation is on three records, two " / "'s should appear in the format.

Both F*w.d* and D*w.d* may be preceded by a multiplier. For example, "3F2.0" in a format is equivalent to "F2.0,F2.0,F2.0". Besides saving space, use of multipliers often makes the format easier to check.

The format must start with "(" and end with ")", but internal sets of parentheses can be set up, typically preceded by a multiplier. This feature is illustrated in the following example.

```
create  in = temp$:[r_fay]w2vplx1.dat
           out = temp$:[r_fay]vplx1.vpl
input   sexw2  agew2 degree  degreex tm8428 racew2 workact
     waitw2  totinc earnings  repf1 - repf100
format  (7x,f1.0,f3.0,3x,f1.0,1x,f1.0,f2.0,2f1.0,f8.5,2f10.1,
         12(/,8f10.7),/,4f10.7)
```

In this example, the input statement defines 110 variables to be read from the input file, and the format statement that follows specifies their locations on a file that has 14 records per observation. The first variable, `sexw2,` is read from position 8 of the first record. The second, `agew2,` occupies positions 9-11. Because 3 more positions are then skipped, `degree` is read from position 15. The remaining variables up through `earnings` are read from the first record. The next part of the format gives a segment to be used 12 times. On each of the 12 uses, the first step is to go to the next record, so that, for example, `repf1` is read from positions 1-10 of the second record. Note that, because the observation corresponds to 14 records, the format accounts for 13 " / "'s.

3.3.6 Programming Hint: INPUT and FORMAT. Errors in the format, or mismatches in length or order between the list of input variables and the format are perhaps the most frequent source of user error during the CREATE step. Unlike spelling and syntactic errors, which VPLX generally spots immediately, errors in the format do not surface until VPLX attempts to use it to read the input file. It is good practice to devote careful attention to the format and input list initially and then to change them as little as possible. It is usually better to read in unneeded variables and drop them from the VPLX file through use of statements such as DROP or KEEP than to modify a working INPUT and FORMAT pair in order to avoid reading extraneous information. (This is an instance where "if it's not broken, don't fix it" applies.)

If a CREATE step produces unusual results, the INPUT and FORMAT statements are almost always suspects. The first recourse is to print a few observations, or several, if necessary, using `/options nprint =` on the INPUT statement, if this was not initially done.

3.3.7 Variable Names. VPLX variable names must begin with a letter or underscore, "_", and contain a combination of 1-12 letters, underscores, and digits. A few variable names are not allowed: AS, BLOCK, BY, CLASS, FOR, IF, INTO, KEY, MEAN, MEANS, MINUS, N, OPTION, OPTIONS, PERCENT, PERCENTS, PERCENT1, PERCENT2, PLUS, PROPORTION, PROPORTIONS, PROPORTION1, TOTAL, TOTALS, and TOTAL1, since these are elements of the syntax. Longer names, such as MEAN_INCOME, that imbed any of these terms are acceptable. (Except for the allowed length of 12 instead of 8, these rules follow those for SAS variable names.) Note that the minus sign ("-") and other special symbols are not allowed as part of a variable name.
VPLX treats upper and lower case spellings as equivalent; for example, `MEAN_INCOME` , `mean_income` , and even `Mean_income` all refer to the same variable.

Some variable names, such as WEIGHT, CLUSTER, STRATUM, can and should be used with reserved meanings assigned by VPLX. For example, WEIGHT identifies a variable normally to be used as a weight for the observations. Nonetheless, in some cases statements in VPLX allow these reserved meanings to be overridden.

Variable names REPW, REPW0, REPW1, REPW2, ..., or REPF, REPF0, REPF1, REPF2, ..., have an entirely reserved meaning, and can only be used to represent replicate weights or factors, respectively. Chapters 12 and 15 explain these uses.

Some other variable names have reserved meanings in specific contexts. For example, when a HADAMARD statement appears in a CREATE or REWEIGHT step, variable names ROW1, ROW2, ... and COEF1, COEF2, ... *etc.*, refer to variables to be used to create either replicate factors or weights. Outside this context, however, the names ROW1, ..., COEF1, *etc.*, do not have a reserved use.

3.3.8 Variable ranges. When names of variables end with one or more digits, the SAS convention of a single "-" may be used to list a string of variable names that increment the ending digits. For example, `month1 - month12` refers to 12 variable names: `month1`, `month2`, `month3`, ..., `month9`, `month10`, `month11`, `and month12`. Note that the ending digits do not have leading 0's. For example, VPLX translates `month01 - month12` into `month01`, `month2`, `month3`, ..., `month 9`, `month10`, `month11`, `and month12`, which may not be the intended extension.

A less frequently used feature, and one which is more difficult to master, employs a double dash "--" between two variable names to indicate a range of previously defined variables in the order in which they have been defined. For example,

```
input  rent gas electric water
constant  12 into c12
multiply  rent -- water by c12
```

uses features to be described in Sections 3.4 and 3.7. The double dash indicates all variables, in this case `rent`, `gas`, `electric`, and `water`, that have been defined between the starting and ending variables given. The double dash "--" may not be used in the INPUT statement or other situations that define variables. The order of variables on the INPUT statement establishes their incoming order, and subsequent statements defining variables effectively add these variables onto the list of defined variables. (Although the "--" feature is offered and documented here for completeness, new users are encouraged to avoid it, unlike the frequently useful and generally straightforward single "-" just described.)

3.4 CONSTANTS

Constants may be copied into real variables. The resulting real variables may subsequently be treated like other real variables. As a simple example:

```
CONSTANTS  1.0 into x1
```

copies the value 1 into `x1`. The variable `x1` may either have previously existed or may be newly defined by this statement.

A more general form of the statement permits simultaneous assignment of constants of a whole group of variables. The general form is:

```
  CONSTANTS   clist1 into vlist1 [ clist2 into vlist2 ]
```

where `clist1` is a list of one or more constants, and `vlist1` is a list of equal length of variable names. The constants in `clist1` may be written with or without decimal points or may be in FORTRAN D*w.d* format. Commas may optionally appear. As an example,

```
  constants  1 2. .37, 17, 17, 17 , -.26713d-07    -.26713d-07
              into c1 - c8   .5 into half
```

In the first part of the statement, 8 values are assigned into 8 variables. Multipliers, such as `3*`, may be used to denote repeated values. Thus, the following statement has the same effect as the previous one:

```
  constants  1 2. .37, 3* 17 , 2 * -.26713d-07 into c1 - c8
                 .5 into half
```

On occasion, it may be advantageous to read the values of the constants from another file. VPLX has an INCLUDE feature that is useful for this purpose and many others.

```
  constants
  include    factors1.dat
                  into f1 - f20
```

where `factors1.dat` is a separate character file with 20 constants in the required form, i.e., free-format, with or without commas, on records ending by position 80, *etc*. VPLX will temporarily read from this file before returning to the original command file to complete processing of the CONSTANTS statement. Section 10.5 further describes INCLUDE and its applications.

Note: VPLX accepts CONSTANT as an alternative spelling of CONSTANTS. In general, any keyword whose English meaning is changed from singular to plural by addition of an "s" may be spelled in either form.

3.5 COPY

The COPY statement copies the current value of a real variable into another. As a simple example:

```
COPY  x1 INTO y1
```

places the value of `x1` into `y1`. The statement does not change `x1`. If `y1` already exists, its current value is overwritten.

The syntax of the general form of the statement is:

```
COPY  varlist1 INTO varlist2 [varlist3 INTO varlist4 ...]
```

where ***`varlist1`*** contains previously defined variables, and ***`varlist2`*** contains the new target variables, *etc*. (The brackets " [" and "] " enclose an optional extension. In general, such brackets are used throughout the documentation to denote optional parts of the syntax.) The variables in ***`varlist2`*** will have the same attributes as those in ***`varlist1`*** at the point the COPY statement appears. Subsequent operations on variables in ***`varlist1`*** will not be automatically reflected in their counterparts in ***`varlist2`***, however. For example,

```
constant  1. into x1
copy   x1 into y1
constant  2. into x1
```

will produce `y1` with the value 1. and `x1` with the value 2. at the end of these three statements.

3.6 IF, ELSE IF, ELSE, END IF

3.6.1 General Features of IF Blocks: VPLX includes a blocked if-then-else construction similar to FORTRAN or to SAS. As a simple example:

```
input  x1  x2
format (2f1.0)
if   x1 (1-3) then
constant 1 into flag1
else if x2 (1-3) then
constant 1 into flag1
else
constant 2 into flag1
end if
```

The logic of this sequence is visually clarified through indentation (Section 3.6.2):

```
input  x1  x2
format (2f1.0)
if   x1 (1-3) then
_   constant 1 into flag1
else if x2 (1-3) then
_   constant 1 into flag1
else
_   constant 2 into flag1
end if
```

After `x1` and `x2` are read from the input file, `x1` is first checked for a value in the range of 1-3. If the value of `x1` falls in the range, then `flag1` is assigned the value 1; otherwise `x2` is now similarly checked and the same action taken, else `flag1` is set to 2. Consequently, at the end of this sequence, `flag1` is assigned either the value 1 or 2.

In general, the construction begins with an IF statement of the form:

```
IF  [(]  vname ( range ) [)] THEN
```

One or more VPLX commands may then follow, which will be executed for the observation only if *vname* falls in *range*. Optionally, one or more statements may then follow:

```
ELSE IF  [(]  vname ( range ) [)] THEN
```

The statements that follow are executed only if the condition is true and if the conditions for all previous IF or ELSE IF statements are false. Whether or not any ELSE IF statements are used, a single:

```
ELSE
```

may follow. The subsequent VPLX statements will be executed if the previous conditions have not been met. In all cases, however, the sequence must be ended by:

```
END IF
```

One END IF must appear for each IF.

VPLX begins the processing of each observation by setting all variables to 0 before reading data from the primary input file or any other operations (*4*). Consequently, if a variable is explicitly defined within an IF block under conditions that are not met for a particular observation, the variable will have the value 0. To modify the initial example,

```
input  x1  x2
format (2f1.0)
if   x1 (1-3) then
_   constant 1 into flag1
else if x2 (1-3) then
_   constant 1 into flag1
end if
```

will result in `flag1` taking either the values 1 or 0.

IF blocks may be imbedded, up to 7 deep. When the level becomes 2 or more deep, VPLX reports the level of each IF, ELSE IF, ELSE, and END IF statement. The example in Section 3.9 illustrates this feature. Again, indentation improves the visual interpretation of imbedded IF blocks:

```
input  x1  x2
format (2f1.0)
if   x1 (1-3) then
_   if x2 (1-3) then
_      constant 1 into flag2
_   end if
_   constant 1 into flag1
else if x2 (1-3) then
_   constant 1 into flag1
end if
```

will result in `flag1` and `flag2` each taking either the values 1 or 0. `Flag1` represents a logical "OR" of the conditions that `x1` falls into (1-3) and `x2` falls into (1-3); `flag2` gives a logical "AND" of the two conditions. The 4*th* line of the example begins an imbedded IF block that ends at the 6*th* line.

The logical restrictions on the use of IF blocks are the same as FORTRAN or SAS. For example, if an IF block at the first level contains an IF block at the second level, the second-level IF block must be concluded with its END IF before an ELSE IF, ELSE, or END IF at the first level appears. The second-level IF block starting at the 4*th* line and ending at the 6*th* in the preceding example is a case in point.

Use of a consistent style of indenting lines to represent the logical nesting should help to write commands complying with these rules.

3.6.2 Indenting Commands. The examples in Section 3.6.1 illustrate indentation. Although VPLX commands have been described as beginning with a key word in position 1, they may optionally be indented by placing a " `_` " in position 1 and then placing the key word in some later position. If the VPLX command is continued on additional lines, however, the continuation lines should begin with blank in position 1.

This feature is especially useful for IF blocks. Section 3.6.1 emphasizes the point that indenting the commands within the scope of an IF block improves the programming clarity.

3.6.3 Range specifications. The syntax for range specifications is similar for IF, ELSE IF, SELECT, CATEGORICAL, CLASS, and other statements. This section provides rules for all of these applications, even though some aspects are not applicable to IF.

A range may represent a single value, such as 1, a bounded range, such as 1-3, or an open ended range, such as low - 1.00. A range specification may include more than one such element, separated by commas. The commas serve to prevent ambiguities:

```
  if   x1 (1-3) then

  if   x1 (1,-3) then
```

the first of the two checks the single range from 1 to 3 while the second checks for the numbers 1 or -3 (*5*). Each single value or range must be separated by commas, whether or not an ambiguity would occur in the specific instance.

In addition, the following may be used in range specifications:

low -	to indicate the lowest possible number, when used as a lower bound in a range. For example, `low - -50` indicates any number less than or equal to -50.
high-	to indicate the highest possible number, when used as an upper bound in a range. For example, `100000 - high` indicates 100,000 or higher.
res -	(for "residual") to represent any value not previously classified. It excludes, however, instances in which a real with missing (Chapter 5) or crossed real (Chapter 6) variable is missing. This range is not generally used with IF statements, but instead with CATEGORICAL, and CLASS statements (*6*).

In interpreting ranges, VPLX implements the SAS convention that any number within 10^{-12} of an integer is that integer.

If the upper end of a range includes a decimal fraction, it is implicitly continued with 9's instead of 0's, e.g. the range (1-1.9) will include any number from $1\text{-}10^{-12}$ to $2\text{-}10^{-12}$ in the interval. Similarly, anticipating the syntax used for CAT and CLASS statements, (1.0-1.09/1.1-1.19/1.2-

1.29) will assign values from $1\text{-}10^{-12}$ to 1.0999999... to the first category, 1.1 to 1.1999999... to the second, and 1.2 to 1.29...... to the third.

Lower ends of ranges are not altered except by the imposition of the rule that any number within 10^{-12} of an integer is that integer. Thus, an integer at the lower end is extended downward by 10^{-12}, as illustrated in the previous example. Other lower ends are not altered.

Warning: Implementation of these rules produces an unusual consequence: the range (low - 0) is actually interpreted by VPLX as (low - .99999999999). If the intended range is effectively any negative number, then this is achieved either with (low - 0.00000000000) or (low - -.1d-11).

3.7 ADD, SUBTRACT, MULTIPLY, DIVIDE

ADD, SUBTRACT, MULTIPLY, and DIVIDE operate on real variables to produce a real outcome. As an example:

```
  add   x1 plus c1
```

This first form stores the sum back into `x1` without changing `c1`. Alternatively, the sum may be placed into a different target:

```
  add   x1 plus c1 into y1
```

These first two examples illustrate a general rule for ADD, SUBTRACT, MULTIPLY, and DIVIDE: either the result is placed into variables explicitly identified by a list following INTO, or the result is stored back into the first operator or set of operators. The second operators, for example, `c1` above, are never changed unless they follow INTO.

ADD may be used on strings of variables as well. One form is:

```
  add   x1 - x6 plus c1
```

which adds `c1` to all six of the variables. Another form adds in pairs:

```
  add   x1 - x6 plus c1 - c6
```

which places into `x1` the sum of `x1` and `c1`, `x2` the sum of `x2` and `c2`, *etc.* Alternatively, the sums may be placed into new variables:

```
  add   x1 - x6 plus c1 - c6 into y1 - y6
```

Another possibility is:

```
  add   x1 - x6 plus c1 into y1 - y6
```

Finally, a form specifies the summation of several variables into a single target:

```
  add   x1 - x6 into sum1
```

Warning: In general, it is better not to include a variable more than once on a list for a given operation, since ambiguities may ensue. For example,

```
  add   x1 plus x1
```

has a straight-forward interpretation, which is to double the value of x1. On the other hand, **avoid**

```
  add   x1 x1 plus x1
```

The implementation in VPLX results in multiplying the initial value by 4, which may not be what the user expects.

The general forms of ADD are:

  `ADD` *`vlist1`* `PLUS` *`vname`*

  `ADD` *`vlist1`* `PLUS` *`vlist2`*

  `ADD` *`vlist1`* `PLUS` *`vname`* `INTO` *`vlist2`*

  `ADD` *`vname`* `PLUS` *`vlist1`* `INTO` *`vlist2`*

  `ADD` *`vlist1`* `PLUS` *`vlist2`* `INTO` *`vlist3`*

  `ADD` *`vlist1`* `INTO` *`vname`*

For the first two forms, the outcome is placed back into *`vlist1`*. For the last form, *`vlist1`* must include two or more variables.

Rules on list lengths: In cases where *`vlist1`* and *`vlist2`* both have multiple elements, the lists must be the same length or VPLX terminates, since the meaning of the intended operation would be ambiguous. Similarly, the length of *`vlist3`* must match the lengths of *`vlist1`* and *`vlist2`* if both lists have multiple elements in the next to last form. These rules apply for MULTIPLY, SUBTRACT, and DIVIDE as well.

The forms for MULTIPLY parallel those for ADD:

```
MULTIPLY  vlist1 BY vname

MULTIPLY  vlist1 BY vlist2

MULTIPLY  vlist1 BY vname INTO vlist2

MULTIPLY  vname BY vlist1 INTO vlist2

MULTIPLY  vlist1 BY vlist2 INTO vlist3

MULTIPLY  vlist1 INTO vname
```

The same rules apply as for ADD, including rules on list lengths.

The available forms for SUBTRACT are:

```
SUBTRACT  vlist1 MINUS vname

SUBTRACT  vlist1 MINUS vlist2

SUBTRACT  vlist1 MINUS vname INTO vlist2

SUBTRACT  vname MINUS vlist1 INTO vlist2

SUBTRACT  vlist1 MINUS vlist2 INTO vlist3
```

Note that SUBTRACT lacks a version of the final form listed for ADD and MULTIPLY. Otherwise, the previous rules apply.

DIVIDE parallels SUBTRACT:

```
DIVIDE  vlist1 BY vname

DIVIDE  vlist1 BY vlist2

DIVIDE  vlist1 BY vname INTO vlist2

DIVIDE  vname BY vlist1 INTO vlist2

DIVIDE  vlist1 BY vlist2 INTO vlist3
```

If the denominator is 0, 0 is the outcome (*7*).

Note that DIVIDE, like SUBTRACT, also lacks a version of the final form listed for ADD and MULTIPLY. Again, previous rules apply otherwise.

3.8 PRINT

The `nprint =` option in the INPUT statement, described in Section 3.3.2, requests printing of the variable values, generally for purposes of checking VPLX applications. The PRINT statement, first available in version 94.06, offers an additional means to check observations. As an example:

```
print    rooms persons cluster
```

PRINT may also use the `nprint =` option.

```
print    rooms persons cluster / option nprint = 3
```

In general, `nprint` may be set to any positive number.

PRINT may be placed inside of an IF block, thus allowing the user to define conditions under which to print the case. The value of `nprint` is compared to the number of observations actually printed, rather than an unconditional count of the number of observations. Consequently, the user can request to see the first 100 cases meeting some unusual condition, regardless of the number of observations that must be screened to find these cases.

3.9 An Example of Real Operations

The following example intersperses several comments describing operations on real variables.

```
comment  EXAM11

comment  This example starts from the data similar to EXAM4 but adds
         an additional variable TENURE.

create  in = exampl11.dat  out = exampl11.vpl

input    rooms persons cluster tenure / option nprint = 3

      4 variables are specified

format   (4f2.0)
```

```
comment  The input data set contains
 5 7 1 2
 6 8 2 2
 5 2 3 1
 4 1 4 2
 8 4 5 1
 8 2 6 1

comment  Create an individual-level ratio of rooms to persons

divide  rooms by persons into proom_indiv                             #1

comment  For purposes of illustration, use the IF construction to
     obtain totals for renters and owners.

if  tenure (2) then                                                   #2

_   copy rooms persons into rooms_rent persons_rent

_   constant 1 into rent_count

_   comment  print cases where the number of rooms is 2.0 or more times
       number of persons, for renters only

_   if  proom_indiv (2 - high ) then                                  #3
    (If level: 2)

_       print  rooms persons cluster
    *** PRINT request    1

_   end if
    (If level: 2)

else  if tenure (1,3) then

_   copy rooms persons into rooms_own persons_own

_   constant 1 into own_count

end if

comment  unconditionally print variables to here.

print  rooms persons cluster tenure proom_indiv                       #4
       rooms_rent persons_rent rent_count
       rooms_own persons_own own_count

    *** PRINT request    2

    (Simple) jackknife replication assumed

    Size of block   1  =            11

    Total size of tally matrix =     11

    Unnamed scratch file opened on unit 13

    Unnamed scratch file opened on unit 14
```

```
**** End of CREATE specification/beginning of execution

Observation     1 from unit 12
     rooms             5.0000          persons             7.0000
     cluster           1.0000          tenure              2.0000
PRINT request    2
     rooms                     5.000000
     persons                   7.000000
     cluster                   1.000000
     tenure                    2.000000
     proom_indiv                .714286
     rooms_rent                5.000000
     persons_rent              7.000000
     rent_count                1.000000
     rooms_own                  .000000
     persons_own                .000000
     own_count                  .000000

Observation     2 from unit 12
     rooms             6.0000          persons             8.0000
     cluster           2.0000          tenure              2.0000
PRINT request    2
     rooms                     6.000000
     persons                   8.000000
     cluster                   2.000000
     tenure                    2.000000
     proom_indiv                .750000
     rooms_rent                6.000000
     persons_rent              8.000000
     rent_count                1.000000
     rooms_own                  .000000
     persons_own                .000000
     own_count                  .000000

Observation     3 from unit 12
     rooms             5.0000          persons             2.0000
     cluster           3.0000          tenure              1.0000
(Printing discontinued on unit 12)                                       #5
PRINT request    2
     rooms                     5.000000
     persons                   2.000000
     cluster                   3.000000
     tenure                    1.000000
     proom_indiv               2.500000
     rooms_rent                 .000000
     persons_rent               .000000
     rent_count                 .000000
     rooms_own                 5.000000
     persons_own               2.000000
     own_count                 1.000000
PRINT request    1
     rooms                     4.000000
     persons                   1.000000
     cluster                   4.000000
PRINT request    2
     rooms                     4.000000
     persons                   1.000000
     cluster                   4.000000
     tenure                    2.000000
```

```
proom_indiv            4.000000
rooms_rent             4.000000
persons_rent           1.000000
rent_count             1.000000
rooms_own               .000000
persons_own             .000000
own_count               .000000
```

Exhibit 3.2 Example illustrating several forms of operations on real variables. The printing of the last 2 observations is not shown. Instances of IF blocks, DIVIDE, COPY, CONSTANT, and PRINT appear.

The INPUT statement includes a request for printing of the first 3 observations. Generally, printing from INPUT statements displays two columns of variables, and is labeled "Observation *n* from unit *u*," where *n* and *u* represent specific values. Printing from PRINT statements requires one line per variable and is labeled "PRINT request *r*."

The divide statement at #1 computes a ratio, `proom_indiv` , of rooms to persons at the household level.

The IF block, beginning at #2, distinguishes renters from owners and prepares separate variables for each subset. There is an imbedded IF block, beginning at #3, to display the basic characteristics of any renter with `proom_indiv` at 2 or more. VPLX identifies the IF block as level 2, since it is inside a level-1 IF block. VPLX also labels the PRINT request as 1, to distinguish it from any other requests that may follow.

The PRINT request at #4 prints each observation, although Exhibit 3.2 ends the resulting display after the fourth observation for the sake of brevity. VPLX numbers this PRINT request as 2.

After summarizing information about the replication method, the output from the requests follow. For the first three observations, the printing from the INPUT statement appears first, followed by the PRINTing of the longer list of variables from request 2. At #5, INPUT notes that it will cease to print observations, since it has printed the requested `nprint` = 3 observations.

The fourth observation is the only one in the data set to trigger the PRINT request 1, since it is the only observation that meets the criteria for `tenure` and `proom_indiv` . PRINT request 2 again follows.

3.10 LINKing FILES

3.10.1 LINK There are many circumstances in which the information for an observation must be assembled from multiple files. Other languages such as SAS and FORTRAN are available to do so. VPLX does not currently address all such situations, but it is able to handle some important special cases.

The first requirement is that the IN= file in the CREATE statement must represent all potential observations. Unfortunately, this requirement excludes some practical applications, including instances in which one might want to interleave observations from different files. If the condition is met, however, VPLX is then able to match to secondary files, each in any of the following three ways:

1) A simple one-to-one match, in which the same number of records are on the primary and secondary file.

2) A conditional match, in which information available from the primary file and any previous secondary files is sufficient to determine whether a record should be read from the secondary file in question. An INPUT statement may appear within an IF block in this case.

3) A keyed match, in which the secondary file is matched to the primary file on the basis of keys. The primary and secondary file must be in sort by these keys, and this sort must not conflict with the sort required by the replication option, e.g., stratum and cluster for the stratified jackknife. For a given set of keys, there can be only one record on the secondary file. Under keyed matching, VPLX checks that one or more of the keyed variables change from the previous record for each record from the secondary file. The INPUT statement may appear unconditionally or within an IF block.

The LINK is of the form:

```
LINK  fname
```

This statement declares a secondary file. It must be followed by a single INPUT and FORMAT statement. The placement of LINK in the series of commands does not initiate reading from this file, rather the placement of the INPUT statement determines when and under what conditions the file should be read (*8*).

The first LINK statement in a CREATE step may only appear after the INPUT and FORMAT statements for the primary IN= input file, to avoid ambiguity. Similarly, the INPUT and FORMAT must be established for a linked file before another LINK statement may appear.

VPLX assigns a FORTRAN unit number to each linked file. VPLX uses this unit number as an identifier to report the number of records read and in any warning or error messages. Currently, VPLX does not retain the original file names for this purpose.

FORMATs must be established for each linked file in the same way as the primary file.

3.10.2 INPUT Under LINK

Unkeyed linking: Under the first or second ways listed to link files in Section 3.10.1, the INPUT statement has the same syntax as previously described in Sections 3.3.1 and 3.3.2, including the options to print or restrict the number of observations. The INPUT statement must list only new variables, that is, variables that have not yet been defined. If INPUT appears within an IF block, the general rule of Section 3.6.1 applies: the variables on the INPUT list will be initialized to 0 and will take new values only if the INPUT statement is executed.

Programming hint: Exactly 1 INPUT statement should appear. If the conditions under which the file should be read are complex, for example, if the file should be read if either of two variables take a specific value, then an approach is to create a flag and to use IF blocks to assign it a value under all circumstances requiring the file to be read. One may then use a simple IF block to check the value of the flag and use a single INPUT statement within the block (*9*).

Keyed linking: The third way listed in Section 3.10.1 provides for a keyed match of the files. This situation is reflected in the syntax for the INPUT statement:

```
  INPUT  vlist / KEY keylist
```

or

```
  INPUT  vlist / KEY keylist /OPTIONS options
```

or

```
  INPUT  vlist /OPTIONS options / KEY keylist
```

where:

a) `vlist` and `keylist` are both lists of variables;

b) the variables in `keylist` are included in `vlist`;

c) the variables in `keylist` are previously defined;

d) the file specified by LINK is sorted by the variables in `keylist`;

e) the records in the file are uniquely identified by the variables in `keylist,` that is, no two records have identical keys; and

f) the variables in `vlist` but not in `keylist` have not yet been defined.

VPLX checks that all of these conditions hold, including enforcing d) and e) as it reads the secondary file. These conditions allow VPLX to unambiguously match many observations from the primary file to single records from the secondary file. For example, the primary file may represent persons and the secondary file may have cluster-level sampling information.

If the match fails (including an end-of-file on the secondary file before the primary file), then VPLX prints a message and terminates.

Unmatched records on the secondary file are discarded and not treated as an error. For example, the secondary file may include cluster-level sampling information for clusters that yielded no interviewed cases, and it is not necessary to remove such clusters from the secondary file. An end-of-file on the primary file before the end on the secondary file is similarly not an error.

Providing a means to link one record on the secondary file to many records on the primary file is only one of the advantages of keyed linkage. Another is security: use of keyed linkage based on unique identifiers, if they are available, is a means to assure that the files have been correctly matched.

3.11 Examples of LINKing Files

Two relatively self-explanatory examples illustrate linking. For simplicity, the TRANSFORM step does not appear. In the first example, the estimates and standard errors from EXAM5 are reproduced by linking information from two other files:

```
comment   EXAM12

comment   This example replicates EXAM5 but reads the data from
          different files.

create  in = example1.dat  out = example1.vpl

input    cluster

      1 variables are specified

format   (4x,f2.0)
```

```
comment  The incoming data, on the file example1.dat, are:
 5 7 1
 6 8 2
 5 2 3
 4 1 4
 8 4 5
 8 2 6

comment  The first link is through way 1, i.e., a simple 1-to-1

link   example7.dat
    Assigned to unit 19                                                         #1

comment   EXAMPLE7.DAT contains:
 5 7 1 1 1 2 2 0 0
 6 8 2 1 1 0 0 2 2
 5 2 3 2 1 2 0 2 0
 4 1 4 2 1 0 2 0 2
 8 4 5 3 1 2 0 0 2
 8 2 6 3 1 0 2 2 0

input  rooms persons

      2 variables are specified

format (2f2.0)


comment  The second link is through way 3, i.e., keyed link

link   example5.dat
    Assigned to unit 18                                                         #2


comment   EXAMPLE5.DAT contains the following data:
 5 7 1 1
 6 8 2 1
 5 2 3 2
 4 1 4 2
 8 4 5 3
 8 2 6 3

input  cluster stratum / key cluster

      2 variables are specified

format (4x,2f2.0)

    Stratified jackknife replication assumed

    Size of block   1  =              3

    Total size of tally matrix =      3

    Unnamed scratch file opened on unit 13

    Unnamed scratch file opened on unit 14

    Unnamed scratch file opened on unit 15
```

```
**** End of CREATE specification/beginning of execution

    End of primary input file after obs #       6                                #3

    End on unit 19 after obs #                  6

    End on unit 18 after obs #                  6

         3 strata observed on incoming file


display

list     rooms  persons  total(rooms persons)


                                                  Estimate      Standard error


rooms                   :  MEAN                     6.0000               .2357

persons                 :  MEAN                     4.0000               .4082

rooms                   :  TOTAL                   36.0000              1.4142

persons                 :  TOTAL                   24.0000              2.4495
```

Exhibit 3.3 Example illustrating linking multiple files. Two files are linked: one by simple 1-to-1 matching, and one through keyed linking.

As noted in Section 3.10.1, the messages at #1 and #2 give the FORTRAN unit numbers to which the files are assigned, which VPLX later references at #3 in reporting the number of records read from each.

Note that VPLX successfully carries out the stratified jackknife option, even though the stratum code is read from the third file. Note that the files are sorted by `cluster` , which increases from 1 to 6, permitting a keyed link on `cluster` .

The second example links a shorter file, within an IF block, to read in data only for owners:

```
comment  EXAM13

comment  This example builds on EXAM11, which first introduced
         tenure.  An additional file is linked for owners

create  in = exampl11.dat  out = exampl11.vpl

input    rooms persons cluster tenure

      4 variables are specified
```

```
format   (4f2.0)

comment  The input data set contains
 5 7 1 2
 6 8 2 2
 5 2 3 1
 4 1 4 2
 8 4 5 1
 8 2 6 1

if tenure (1) then

_   link     exampl17.dat
    Assigned to unit 19

_   comment    The input data set contains:                                          #1
       3 100000  75000
       5 150000 100000
       6 100000      0

_   input   assessment  mortgage

      2 variables are specified

_   format (2x,2f7.0)

end if

print tenure assessment mortgage
    *** PRINT request    1

    (Simple) jackknife replication assumed

    Size of block   1  =              6

    Total size of tally matrix =      6

    Unnamed scratch file opened on unit 13

    Unnamed scratch file opened on unit 14

**** End of CREATE specification/beginning of execution

PRINT request    1
     tenure                  2.000000
     assessment               .000000
     mortgage                 .000000
PRINT request    1
     tenure                  2.000000
     assessment               .000000
     mortgage                 .000000
PRINT request    1
     tenure                  1.000000
     assessment         100000.000000
     mortgage            75000.000000
PRINT request    1
     tenure                  2.000000
     assessment               .000000
     mortgage                 .000000
```

```
PRINT request    1
     tenure                  1.000000
     assessment         150000.000000
     mortgage           100000.000000
PRINT request    1
     tenure                  1.000000
     assessment         100000.000000
     mortgage                 .000000


    End of primary input file after obs #       6                                  #2

    End on unit 19 after obs #                  3
```

Exhibit 3.4 Example illustrating linking files with different numbers of observations. The matching is controlled by placing the INPUT statement within an IF block, so that values from the linked file are read only for owners.

In this example, a separate data file, shown at #1, has been prepared only for owners. The file contains the cluster number that the format skips in this example. The approach taken in this example places the INPUT statement within an IF block and reads from the file only for owners. Since the linkage is unkeyed, a new record from the secondary file is read each time the INPUT statement is reached. PRINT is used here to show how the owners' data are correctly linked with the intended records. The results from PRINT show that `assessment` and `mortgage` are set to 0 for each renter.

The report at #2 gives the number of records read from each file. Again, as noted in Section 3.10.1, VPLX identifies secondary files by unit number.

3.12 Selecting Individual Observations: SELECT IF

As noted in Section 3.1, the statements described in Sections 3.12-3.15 are of a different character than the preceding sections. These last four sections describe features that influence the contents of the resulting VPLX file. In the previous sections, it was possible to illustrate the effects of different statements by PRINTing the results, but the effects of this last group of statements to be considered in this chapter are not directly evident in this way. Because the statements affect the contents of the outgoing file, there effect is not immediately evident in terms of the values of the variables during the processing of the observation.

The SELECT or SELECT IF statement may be used to determine what observations are included in the file. This purpose is different from IF, since IF enables statements to be executed conditionally. SELECT does not affect the execution of other statements. Instead, SELECT determines whether the observation is to be included in the final results. The form is:

```
SELECT [IF]  vname (range)
```

The optional IF included in the syntax improves the clarity of the meaning of the statement. The specification by `range` should identify values of the variable for which the observation should be included in the analysis. Section 3.6.3 specifies the syntax for `range`, which is the same as used by IF. If the variable is not within the desired range, the case is dropped from all tallies, regardless of the order of commands that precede or follow. In other words, SELECT has a global effect on the inclusion of cases. For example,

```
select if  degree_stat (1-3,8)
```

will include the observation in the analysis only if `degree_stat` takes a value between 1 and 3, or 8.

If two or more SELECT statements are included, then each such condition must be met for inclusion of the observation.

Even if an observation is excluded from the output file through failing a SELECT IF condition, CREATE will continue to process the case to the end, including effects of LINK, INPUT, PRINT, *etc*.

Programming hint: It is advantageous to include SELECT statements at the end of the CREATE specification. This placement emphasizes that other commands will be executed regardless of the outcome of the conditions. It also more clearly represents the role of SELECT in affecting the final disposition of the observation.

3.13 Weighting Cases: WEIGHT, `weight`, and UNWEIGHTED

The statement

```
WEIGHT   vname
```

may be used to declare any defined variable as a weight. In the absence of this explicit command, VPLX will assume that a variable with the name `weight` is to be used as a weight.

It is important to identify the weight to VPLX as soon as possible. For example, it is good practice to include the WEIGHT statement just after the weight has been read from a file or its value defined by calculation. Some features of CREATE, but particularly REWEIGHT, require early identification.

An obscure but possibly useful option: It is also possible to include a variable with the name `weight` but nonetheless carry out an unweighted analysis with the statement:

```
UNWEIGHTED
```

This last statement is only required for unweighted analysis if, for some reason, there is a variable with the name `weight` present among the defined variables. For example, one might conduct both a weighted analysis based on `weight` and then run a separate unweighted analysis, using UNWEIGHTED, without changing an INPUT statement that contained `weight`.

3.14 Controlling the Contents of the Output: DROP and KEEP.

By default, VPLX will retain all defined variables on the VPLX file. Optionally, the user may restrict the contents of the VPLX file by either a single DROP or KEEP statement. The form of the statement is:

```
DROP   vlist

KEEP   vlist
```

Only one of these two statements may appear in the CREATE stop. Dropped variables will be omitted from the output file, whereas only those analytic variables in the KEEP statement will be included in the file. Either statement is evaluated with respect to the variables defined at the end of the complete CREATE specification, in other words, the variables in *`vlist`* do not need to be defined at the time the statement appears but must be defined by the end of the complete CREATE specification. (Any use of the "--" syntax for variable ranges, however, requires that the variables be defined by the time of reference.) Neither statement has any effect on weights, class variables, replicate weights or factors, or survey identifiers required for the replication method; these variables will be handled in the same way regardless of whether they appear on a KEEP or DROP list. Any defined variable remains available for calculation during the CREATE step, regardless of its status on the outgoing file.

3.15 Labels for Variables: LABEL

Variables may be assigned labels of up to 24 characters for use in displays. The form of this statement is:

```
LABEL    vlist1  'label1a',...
           vlist2  'label2a', ...
```

where the number of variables in each variable list matches the number of labels that follow. A simple form of this syntax is to alternate single variable names with corresponding labels.

Alternatively, the keyword may be LABELS. For example:

```
  labels   sex  'Sex' total_earn  earn_cat
             'Total earnings in 1985' 'Total earnings in 1985'
```

Every variable is initially labelled with its variable name, but a LABEL statement overrides this default.

NOTES

1. .CRD is an abbreviation of CARD. In the early 1970's, input to programs was often through 80-column computer punch cards. Rubber bands, receptacles for cards, *etc*., were standard equipment of the era. The author finds the abbreviation a useful mnemonic, but the user is free to chose another convention.
2. The VPLX statements that do not allow extensions are restricted to those in Chapter 13. In each case, such as ECHO OFF, the rest of the statement will conveniently fit on one line. Many VPLX statements statements, such as INPUT, could require more that a single line to complete, and VPLX allows multi-line continuations in all such cases. The examples throughout the documentation frequently employ continuations.
3. This limit is on the number of characters of the specified format itself, not the total number of characters in the input records that the format describes. Depending on the application, a 50-character format may describe thousands of character positions on the IN= data file. If the constraint proves a hinderance, however, it is possible to increase the maximum length by changing a parameter in the FORTRAN source.
4. More precisely, the processing of each observation begins by putting 0's in the values of all variables in situations in which any new variables are introduced within an IF block. If new variables are not introduced within an IF block, then initializing variables to 0 is unnecessary and is skipped. For simplicity, however, the reader may simply wish to consider all variables set to 0 at the beginning of processing an observation.
5. This statement is generally correct, but a more precise statement requires the rules on decimal extensions that follow. The range 1-3 actually becomes the range $1\text{-}10^{-12}$ to $4\text{-}10^{-12}$. Although the range comes close to 4, it does not include it, and any number read as 4 from a file will not satisfy the condition. Similarly, (1,-3) checks for $1\text{-}10^{-12}$ to $1+10^{-12}$ and $-3\text{-}10^{-12}$ to $-3+10^{-12}$.
6. Both real with missing variables and crossed real variables may be missing. If X1 is missing, then
   ```
   IF X1 (res) THEN
   ```
   will not be satisfied. Hence, the following two statements
   ```
   IF X1 (res)  THEN
   ELSE
   ```
   can introduce subsequent statements that should only be executed when X1 is missing, (followed, of course, by an END IF statement.) If X1 is a real variable, however, the statement
   ```
   IF X1 (res)  THEN
   ```
   serves no purpose since it will always be satisfied.
7. If the 0 outcome is unsatisfactory, there are two alternatives. One is to check whether the divisor is 0 with an IF statement, and to do something special, such as to set a flag, in this case. The other option is to use DIVIDE_MS in Chapter 5.

8. Users familiar with SAS may wish to check further comments on this point in Section B.2. SAS INFILE statements may be meaningfully interpreted within IF blocks, but in VPLX the LINK statement identifies the next file to read, regardless of placement within an IF block. In VPLX, it is the placement of the input statement that determines when data are read.
9. If one wants to read a file only if X1 is 1 or X2 is between 2 and 6, then

```
IF X1 (1) THEN
_   CONSTANT  1 INTO FLAG1
ELSE IF X2 (2-6) THEN
_   CONSTANT  1 INTO FLAG1
END IF
IF FLAG1 (1) THEN
_    LINK  FILE1
_    INPUT  V1 - V3
_    FORMAT (3F1.0)
END IF
```